# OOSD Assignment 1

(Group Name: ‘The High Distinctions’)

## Analysis / Gameplay & System Design:

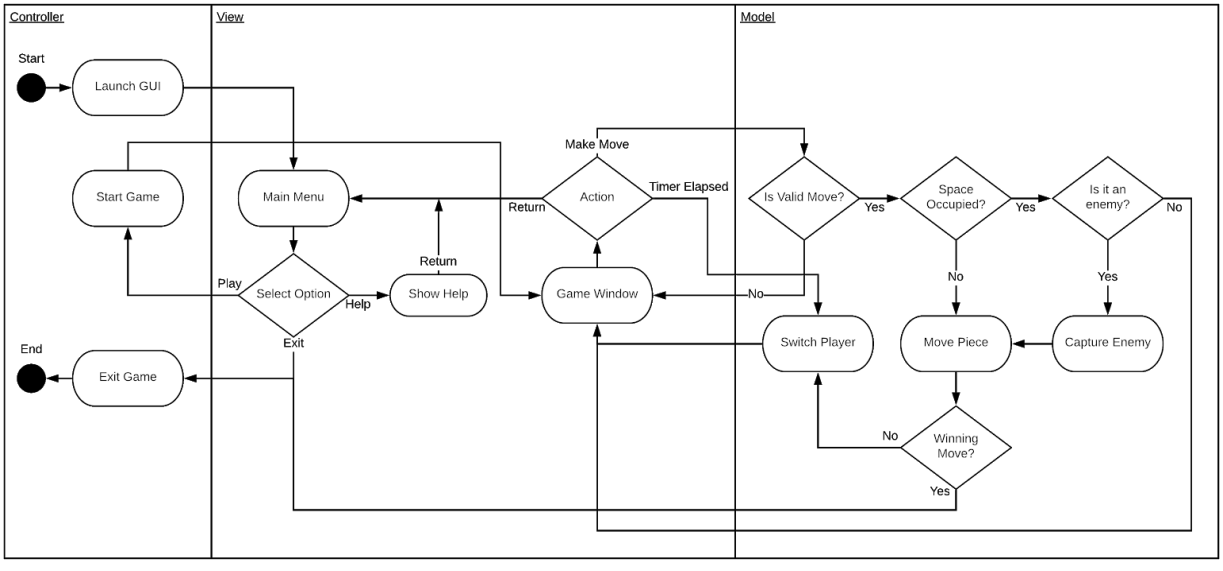
There are 2 teams in this game: The merchants and the bandits. The goal of the merchant is to get to the top side of the board and escape the bandits. The goal of the bandits is to capture the gold merchant for his goods, before he escapes.

The game always starts with the Merchant team. Each team takes turns moving one piece of their choice. Each unit has a different Move value and Power value. The Move value determines how many squares a unit is able to move to. The Power value determines the strength of each piece. For example, a unit with Power 3 is able to capture a unit of Power 1, however, a unit of Power 1 is unable to capture a unit of Power 2 or Power 3. If 2 Power 3 units fight, both will be captured.

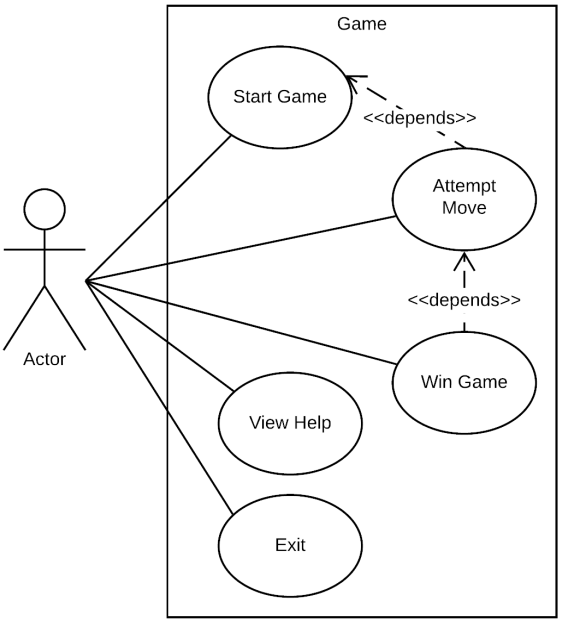
| **Name** | **Move** | **Power** | **Ability** |
|---|---|---|---|
| Gold Merchant | 2 | 0 | Carries the gold |
| Guard | 3 | 2 | Is able to take 2 hits |
| Pikeman | 2 | 3 | Can range attack 1 square away |
| Assassin | 4 | 1 | No obstacle penalty |
| Wizard | 2 | 2 | Can range attack 1 square away |
| Evil Knight | 3 | 3 | Instant kills anything (ie kill Guard) |

Table 1: Movement, power and special abilities of the different piece types. NB: Power levels and special abilities have not been implemented in our game yet, this will be introduced in part 2.

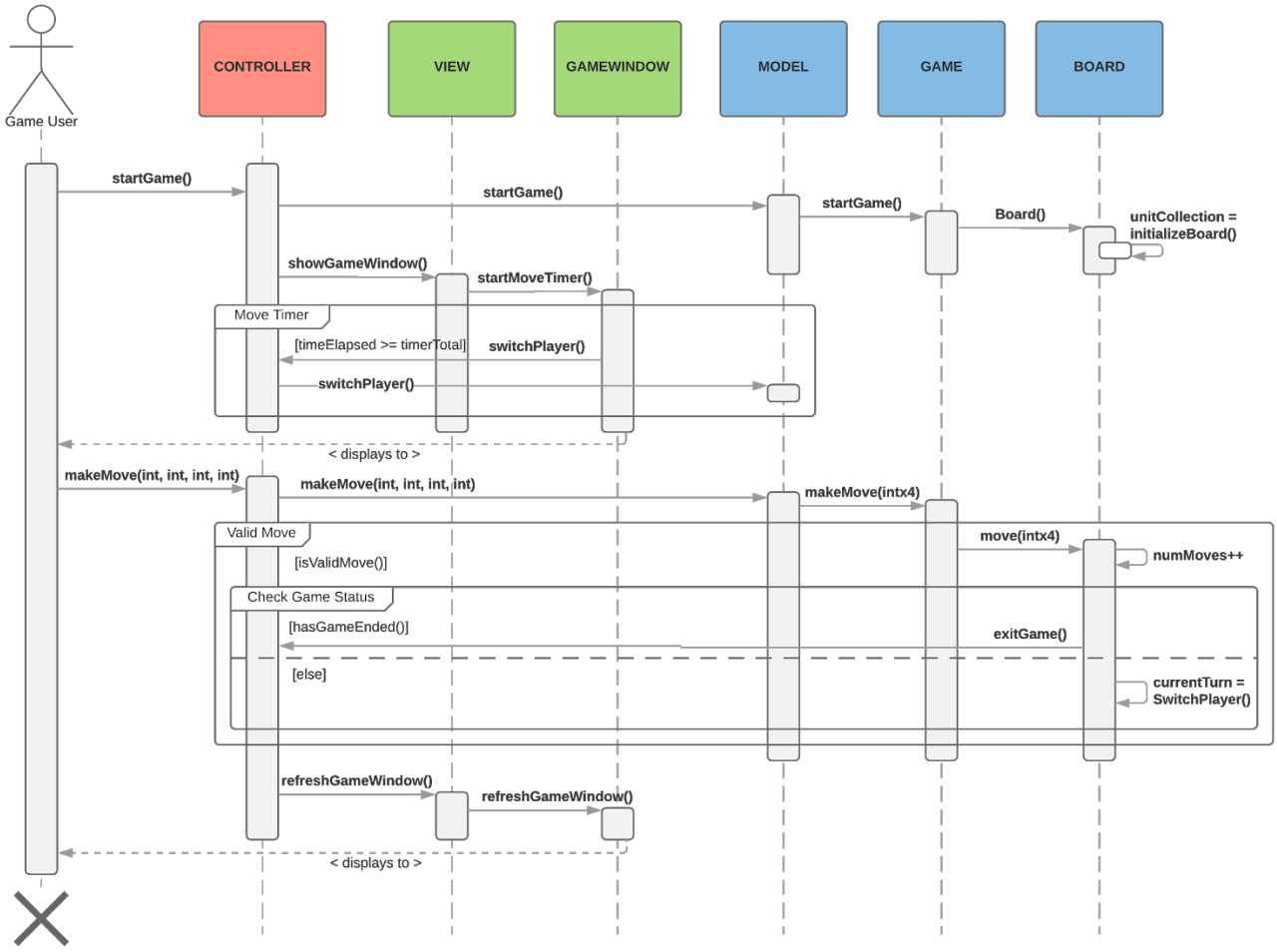
Activity Diagram, shows the basic pathway of logic.



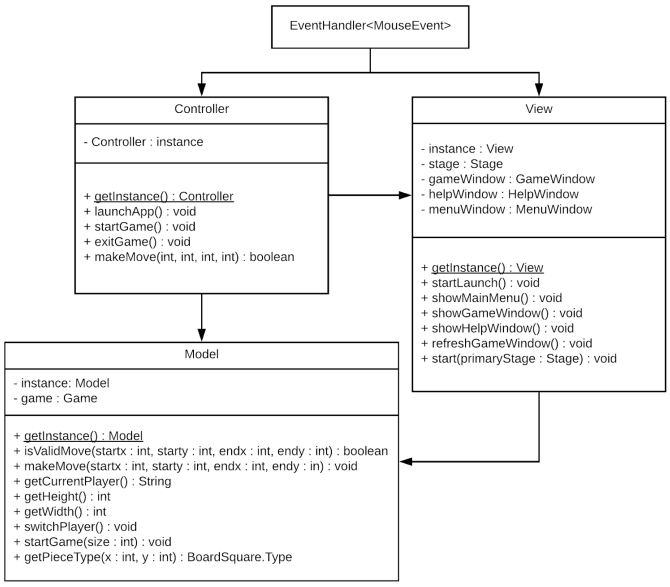
Use Case Diagram, showing available actions to the user

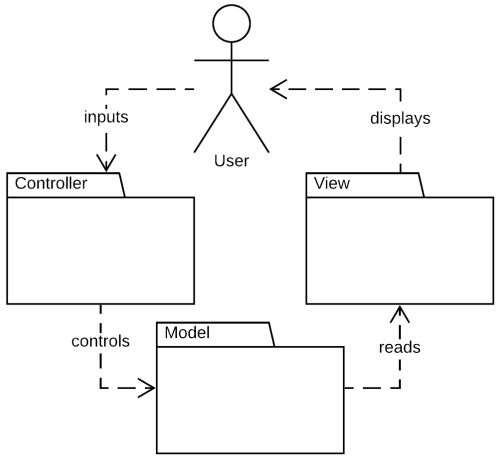



Sequence Diagram, shows a user starting a game, as well as making a move.



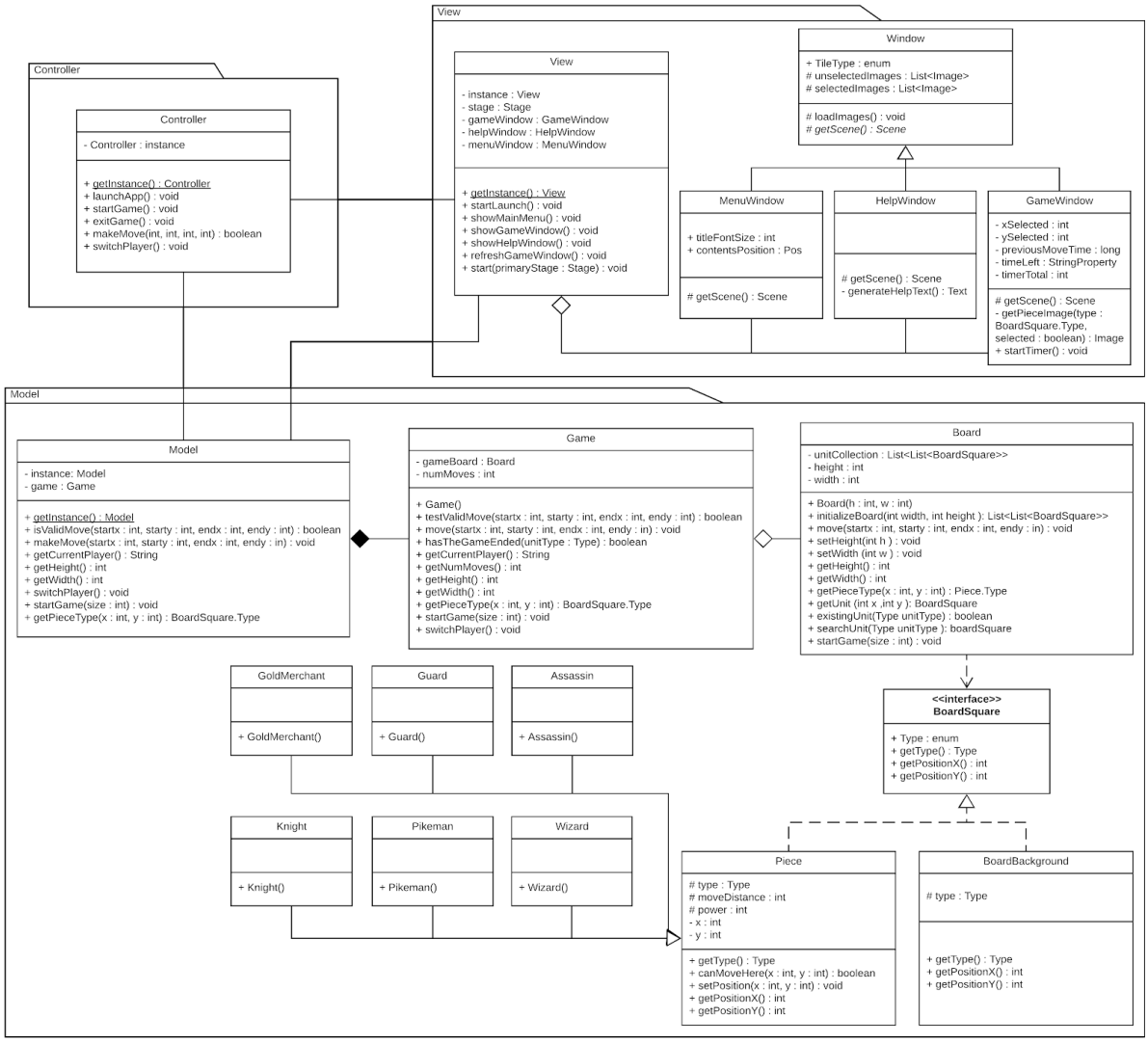
The current system design follows a MVC architectural pattern. This allows for efficient code reuse, and parallel development.



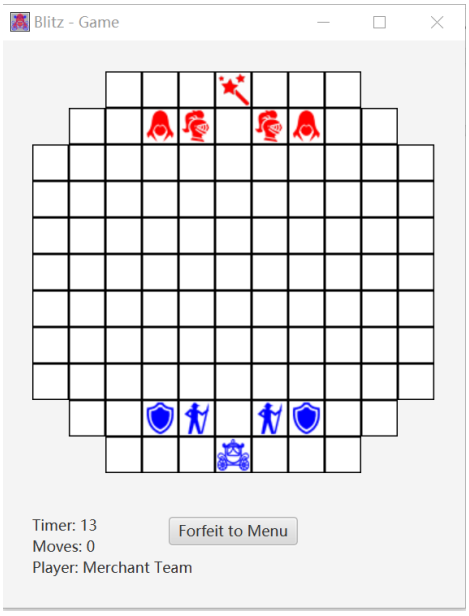

Class Diagram, showing classes categorized by their packages:

---

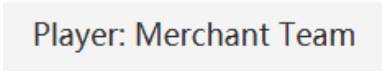


# System Implementation (using javaFX) :

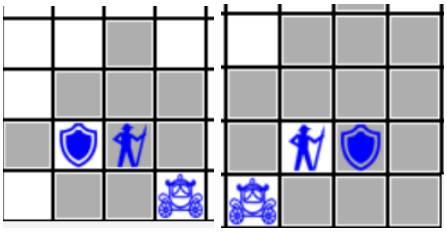
The game board, and the grid of square that hold the pieces:




We show who's teams move it currently is:




Selected pieces have a grey background, and can move to all other grey squares:



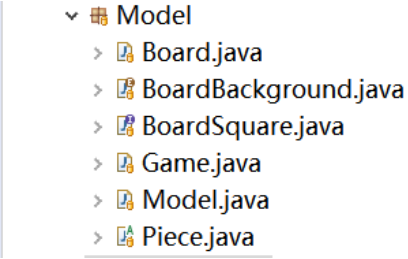
Timer shows remaining time to make your move:

# Non Functional Requirements:

## The use of SOLID and GRASP in the code:

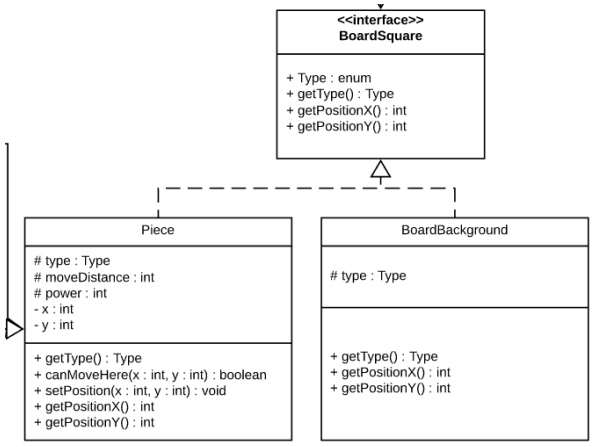
```
@Requires({ "width > 0", "height > 0" })
@Ensures({ "unitCollection != null" })
private List<List<BoardSquare>> initializeBoard(int width, int height) {
    // Create 2D array list
    unitCollection = new ArrayList<List<BoardSquare>>(height);
    // Default unit
    BoardSquare boardSquare;
    for (int h = 1; h <= height; h++) {
        List<BoardSquare> row = new ArrayList<BoardSquare>();
        for (int w = 1; w <= width; w++) {
            if ((h == 1 && w == 1) || (h == 1 && w == 2) || (h == 1 && w == height - 1) || (h == 1 && w == height)
                    || (h == 2 && w == 1) || (h == 2 && w == height) || (h == height - 1 && w == 1)
                    || (h == height - 1 && w == height) || (h == height && w == 1) || (h == height && w == 2)
                    || (h == height && w == height - 1) || (h == height && w == height)) {
                boardSquare = BoardBackground.OBSTACLE;
            } else if (h == height && w == (width / 2) + 1) {
                boardSquare = new GoldMerchant();
            } else if (h == height - 1 && w == width / 2) {
                boardSquare = new Pikeman();
            } else if (h == height - 1 && w == width / 2 + 2) {
                boardSquare = new Pikeman();
            } else if (h == height - 1 && w == width / 2 - 1) {
                boardSquare = new Guard();
            } else if (h == height - 1 && w == width / 2 + 3) {
                boardSquare = new Guard();
            } else if (h == 1 && w == (width / 2) + 1) {
                boardSquare = new Wizard();
            } else if (h == 2 && w == width / 2) {
                boardSquare = new Knight();
            } else if (h == 2 && w == width / 2 + 2) {
                boardSquare = new Knight();
            } else if (h == 2 && w == width / 2 - 1) {
                boardSquare = new Assassin();
            } else if (h == 2 && w == width / 2 + 3) {
                boardSquare = new Assassin();
            } else
                boardSquare = BoardBackground.EMPTY;

            if (boardSquare instanceof Piece) {
                ((Piece) boardSquare).setPosition(w - 1, h - 1);
            }

            row.add(boardSquare);
```

**Creator Pattern** : The board calls the function ‘initializeBoard,’ which means the board has a composition relationship with the BoardSquare type. This is similarly applied throughout the code.
**Polymorphism** : The boardsquare doesn't need to mention what specific type it is and thus can be used transparently with any number of new types.
**Open - Close**: The BoardSquare abstract is not closed to modification when extending with another piece type. It is useful for the programmer when we need more different pieces.
**Liskov substitution**: The specific piece type will always work when treated as BoardSquare abstract class.

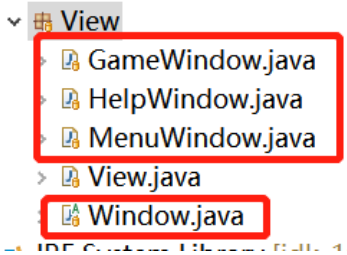



**Single-responsibility** : each class only has one motivation for changing
Board: The collection of Boardsquare, and Boardsquare position management
BoardSquare: The unit of Board, the primary task is determined to position what specific is.
Piece: The abstraction of specific type Boardsquare.
BoardBackground: same as above.

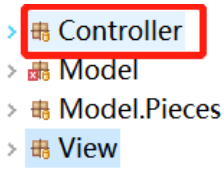
**Pure Fabrication** :The unit of the board does not belong anywhere, and it needs to exist in the application that doesn't announce itself as an obvious class or real-world object, so rather than enforce that unit into board class where it doesn't belong, we fabricate a new class




**Interface segregation** : break up the Boardsquare interface to deal with different implement, the abstract class Piece will generate the specific piece, and the BoardBackground will load the token instead of Boardsquare.

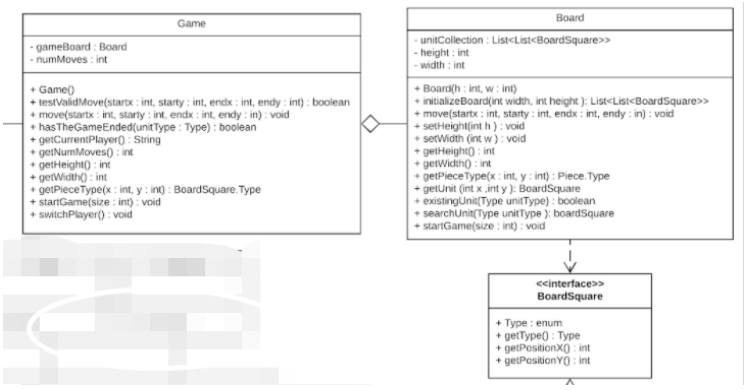



**Protected variations** : Using Window interface so that changes and variation of others view have the minimum impact on what already exists.




**Controller** : Separating the business object and user interface object, it makes the working individual.

**Indirection** :The model not directly work on user interface.

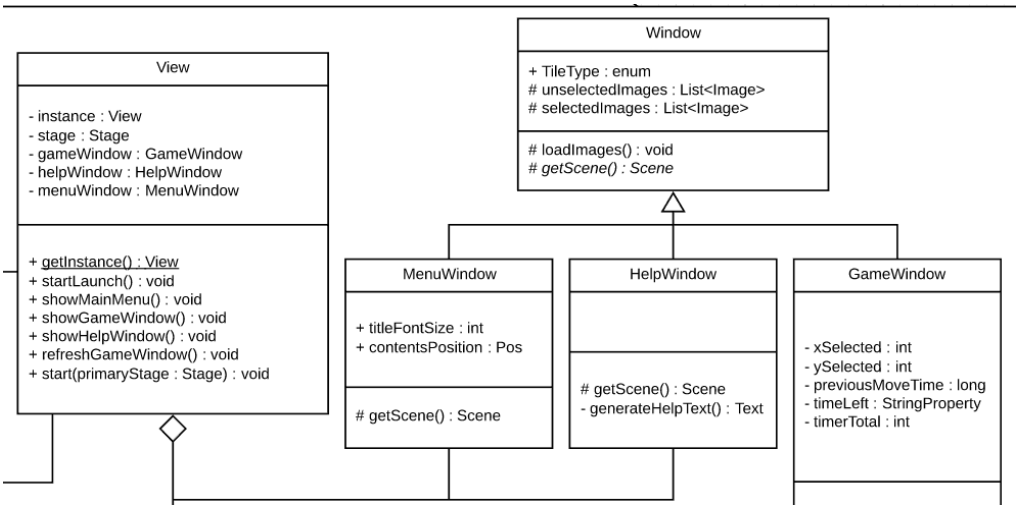



**Dependency inversion** : Add the layer of abstraction ‘BoardSquare’, it avoids the high-level objects know how the low-level work. Using dependency inversion is not directly tying concrete

objects and making them deal with abstractions to minimize dependencies between objects, it allows substitution, flexibility going forward, being able to replace and extend without changing the high-level class at all.

```
@Requires({ "startx > 0", "starty > 0", "endx > 0", "endy >0 ", "startx  < width ", "starty < height",
        "endx < width ", "endy < height" })
public void move(int startx, int starty, int endx, int endy) {
    BoardSquare unit = unitCollection.get(starty).get(startx);
    ((Piece) unit).setPosition(endx, endy);

    unitCollection.get(endy).set(endx, unit);
    unitCollection.get(starty).set(startx, BoardBackground.EMPTY);
}
```

**High cohesion** : the method should not do many things, it makes the the code easy to read, help isolated the problem going to be.




**Low coupling:** changing the view code doesn't affect others code performance, like above, the different type of window works separately, the view will be treated the same

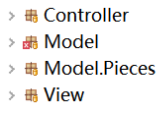
## Evidence of Design by Contract (DbC):

(cofoja 1.3)

```
@Requires({ "startx > 0", "starty > 0", "ednx > 0", "endy >0 ", "startx  < width ", "starty < height",
        "endx < width ", "endy < height" })
public void move(int startx, int starty, int endx, int endy) {
    BoardSquare unit = unitCollection.get(starty).get(startx);
    ((Piece) unit).setPosition(endx, endy);

    unitCollection.get(endy).set(endx, unit);
    unitCollection.get(starty).set(startx, BoardBackground.EMPTY);
}
```

The index of starting move from the user input is must confirmed legally before game operate move function , so all of the index need to be greater than zero, but not exceed the board boundary.

## Separation of Model, View and Controller :




Using named package to store corresponding aspects of code makes working individually easier.